

VVV / Projection / Kinect



Table of Contents

0. Basic Principles	page 6
1. IOBox.....	page 12
2. IOBox II.....	page 14
3. IOBox III.....	page 16
4. DirectX.....	page 18
5. Transforms	page 20
6. Spreads	page 22
7. Animation	page 28
8. Spreads II	page 32
9. Transforms II.....	page 36
10. DirectX II	page 40
11. Shaders.....	page 42
12. Audio response	page 46

VVVV / Projection / Kinect

Author: Elliot Woods

Design: Hwa Young Jung

First Edition, November 2012. Version 1.0











Produced by Manchester Digital Laboratory / Omniversity of Manchester Press.

<http://omniversity.madlab.org.uk/>

©2012 Elliot Woods



Legend

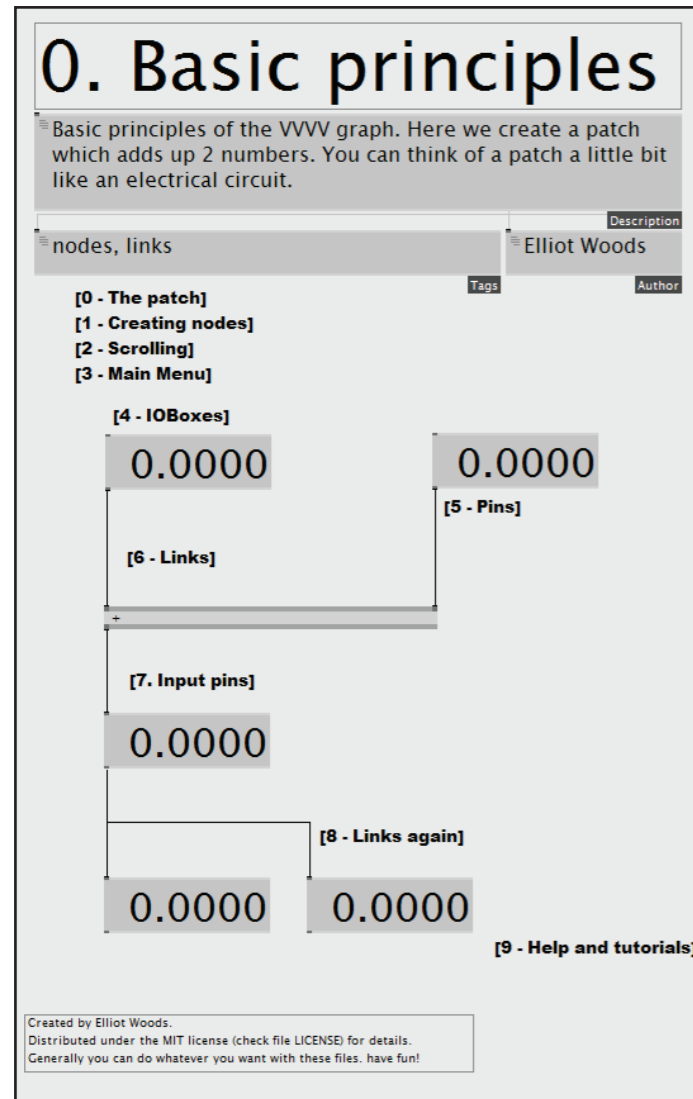
	mouse
	left mouse click
	middle mouse click
	right mouse click
	scroll wheel
	double click right
	node
	pin
	concept
	keystroke

Introduction

This document serves as a reference whilst we work through the examples in the class, and to use as a reference when later reviewing the examples.

The style of this document is terse and accurate, and therefore (as of March 2012) not designed in itself to act as a standalone textbook.

0. Basic Principles



0.0 The patch

When we open VVVV, we get a blank square.

This blank square is a view of a new blank VVVV document, which is called a Patch.

Patches define 'programs' in VVVV. A patch can perform all sorts of tasks. It can act on its own, or can perform a specific function within a bigger patch. When one patch is nested within another, it is called a Subpatch.

The Graph is a term given to everything that VVVV is thinking about. This word can commonly be used interchangeable with the 'Patch', but the Graph is also a more general term which incorporates all running patches and subpatches.

Patches are made entirely out of Nodes and Links.

We denote a Node like this: Node name.

Almost all of the interface for VVVV is handled by nodes, and therefore there are no toolbars, menu bars, etc. to start with. The main exception to this is the Main Menu.

0.1 Creating nodes

Click the mouse in an empty space in the patch window.

This will bring up the NodeBrowser which you can use to create new nodes.

If you start typing, the NodeBrowser will begin searching for new nodes that you can create.

Use the up/down arrow keys on the keyboard to select different nodes within the list. Use the scroll wheel on the mouse to scroll the list. Press **Enter** on the keyboard or click on an item in the list to create the node.

Click on the text box at the top of the NodeBrowser, to browse for nodes by category.

0.2 Scrolling

There are 2 main ways to scroll around the patch in VVVV. You can either:

1. **Drag right mouse button** Move your mouse over an empty area of the patch, hold down your right mouse button and then start moving your mouse. Release the mouse to stop scrolling.

You will notice that the whole patch will move with respect to the window. This is the most common form of scrolling. But **WARNING!** Be careful not to right click on any links in the patch. It's common for beginners to scroll like this with their mouse over a link. Right clicking on a link deletes it!

2. **With the scroll wheel** Safer for busy patches (for the link-deleting reason described above). You can scroll up/down using . You can scroll left/right with **Alt** + and using the scroll wheel. You can scroll super-quick using **Ctrl** + .

0.3 Main Menu


The main menu contains a set of functions. All except a couple of which can be accessed without using the main menu.

To open the main menu, move your mouse over an empty area of the patch and click the middle mouse button. If you don't have a middle mouse button, then use **Space** + .

The main menu performs common tasks such as Open, Close, Save, Quit, Copy, Paste, etc. as well as some VVVV specific functions which may help you debug your patch, direct you to help, speed up your patch.

0.4 IOBoxes

An **IOBox** is a special node. It is the most basic method for inputting data into VVVV (input), and for seeing the data for ourselves (output). We can only edit data inside an IOBox if it has nothing connected to its input.

We can create an **IOBox** by double right clicking on the patch. 

Try changing the value by double clicking on it, and entering a value manually, then press **Enter**

0.5 Pins

Nodes need a way of communicating with other nodes.

The ‘cable’ that connects 2 nodes together is called a Link, and the ‘socket’ where that cable connects to the node is called a Pin.

In this documentation, we demonstrate a pin with the following style: style. This could denote either a specific pin such as **+ (Value)** node’s Input 1, or more casually refer to a general pin, such as the input of an **IOBox**. Output pins are on the bottom of the nodes, Input pins are on the top of the nodes.

An input can only be connected to 1 other node’s output at any time.

An output can be connected to many other nodes’ input at any time.


Move your mouse over a pin to see what value it has (works for both inputs and outputs).



0.6 Links

As mentioned before, a connection between 2 nodes is called a ‘Link’.

A link is made between an input pin (on the top of a node) and an output pin (the bottom of another node).

We can think of the data flowing from the output of one node, down the link, and into the input pin of the ‘downstream’ node.

To make a link, first select a pin to start making the connection by left clicking  on the pin. You must remember to click (and not to drag, especially if you’re used to Max/MSP style linking).

Once you have clicked once and made sure to release the mouse button  , you are now ready to select the target pin for your link. Pins that you are allowed to connect to (i.e pins that you can make a meaningful connection with) are highlighted by becoming graphically larger on the screen. Click  on a target pin to complete the link.

To delete a link. Right click on it, or select it by left clicking on it, and hit **Delete**

0.7 Input pins

Data sent via a link is sent from an output pin and received on an input pin. Input pins are the pins across the top of a node.

Input pins have different names depending on their function. For basic nodes, the main input is called Input. Put data in there and get a result on the output.

There may also be other input pins also called “Input”, e.g. a **+ (Value)** node has by default 2 inputs, Input 1, Input 2 which are the 2 numbers to be added together.

Sometimes it’s appropriate for the input pins to have different names, e.g. a **HSL (Color, Join)** node has 4 inputs: Hue, Saturation, Lightness, Alpha. These are joined together into 1 colour value on the Output.


Input pins can generally store values. If you connect to an input pin, then disconnect, it will store the last value given to it. You can also edit the value of an input pin without connecting anything to it






Sometimes the input pins are attributes for how that node should behave, e.g. A **Renderer (EX9)** node has a Fullscreen pin, which allows you to set that renderer’s window to be fullscreen.


0.8 Links again

1 output can be linked to several inputs

1 input can only be linked to 1 output

Links can be created by either first select input then output, or vice versa. Both times, using 

If you want to make a link from 1 output to multiple inputs, then you can either create them all one by one using  , or you can create them more quickly by starting to create the link from the output pin using the right mouse button  . Now every time you  an input to make a connection, you will still have the output selected and be free to create a new connection to another input using 1 more  without reselecting the output. Right click  in blank space to get rid of the link you are creating..

If you start to make a link by accident, right click  in a blank area to clear.


A common error is to ‘miss’ when trying to make a connection (this is either through being inaccurate with the mouse, or by selecting ^an incompatible to connect to). This means that no link has been made, but you still likely have the link attached to your mouse. Be careful of this, make sure the link is created properly and still exists after you move the mouse away! Otherwise try again.

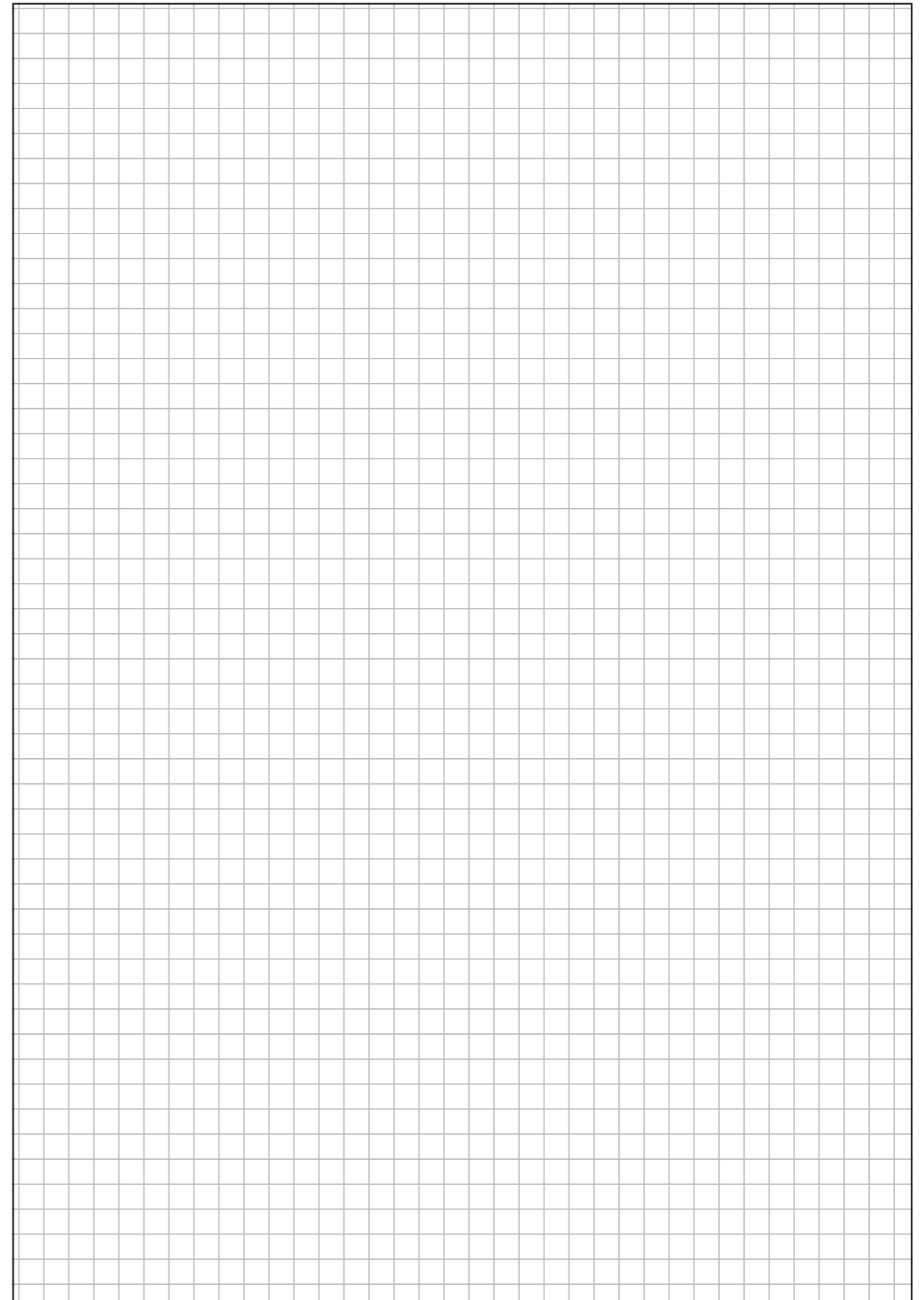
To ‘smarten up’ your links follow these steps:

1. Select them (either lasso a few of them at once by dragging out a selection box around them, or select them one by one with left mouse button).
2. Press **Ctrl** **Y** until you get the style of your choice (Right angles, Curves, Direct)

0.9 Help and tutorials

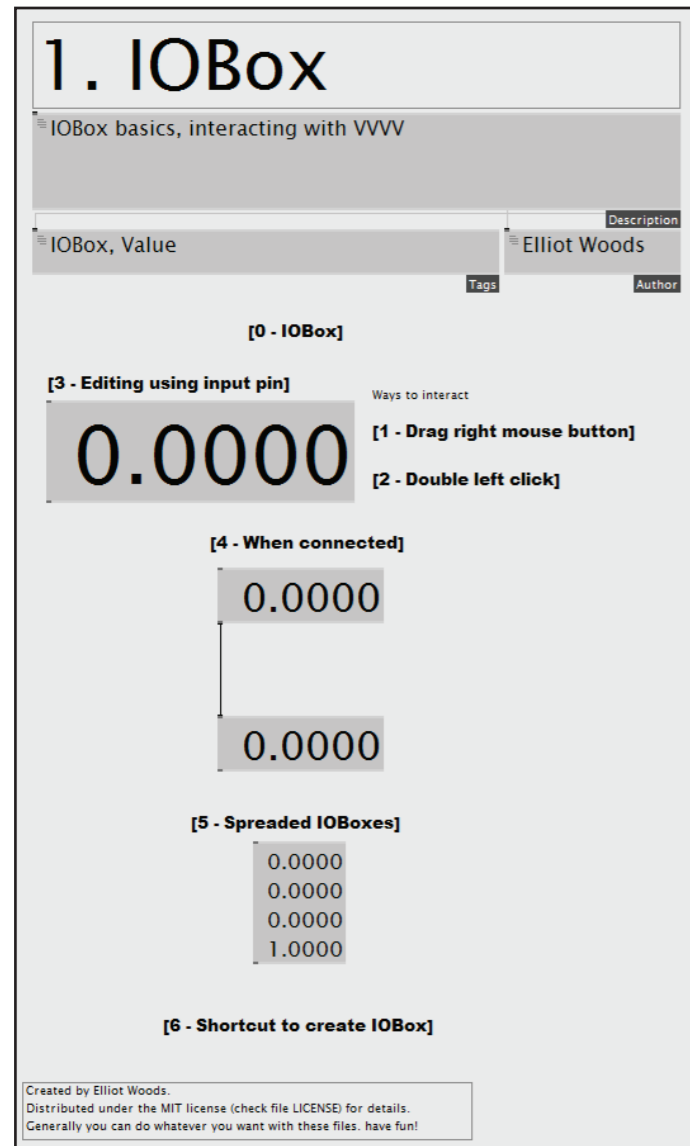
There are some great tutorials to help you out if you're stuck / get curious:

- If you want help about a node, select it with  and press **F1**
- If you want to see examples of what you can do with VVVV, check the **girlpower** folder within the VVVV directory for sample patches
- Make sure to make a user on the *www.vvvv.org website*, and post your queries in the forum
- Check out West's *VVVV tutorial videos on YouTube*



1. IOBox

1.0 IOBox



Open the example patch for this chapter. The most basic input / output mechanism in VVVV is the **IOBox**.


There are different IOBoxes for different types of data (Value, String, Color, Enum, Node).

The most basic type of data is called a Value, which means a real number (note for programmers: this is currently represented by a double precision floating point value).

IOBoxes perform multiple roles:

1. Give opportunities for user input/output
2. Holds data when no input is connected
3. Perform access to inputs/outputs of subpatches (more on that later!)



1.1 Drag right mouse button

Move your mouse cursor over the IOBox without pressing any buttons. Now hold down the right mouse button  and drag up/down to change the value.


To move through values more slowly (more accuracy) hold down either **Ctrl** or **Shift** whilst dragging the right mouse button:

e.g. **Ctrl** + . Hold down both to get even more accuracy **Ctrl** + **Shift** + 


To move through values more quickly hold down **Alt** + **Ctrl** or **Alt** + **Shift** whilst dragging the right mouse button. Hold **Alt** **Ctrl** **Shift** for maximum speed.

In VVVV, generally we use the right mouse button  for interacting with Values and other datatypes. We use the left mouse button  to change the patch itself. You best get comfortable with that right mouse button because you'll be using it a lot!

1.2 Double left click

The alternative way of changing the value is to double click on the IOBox  to change the value by entering it with the keyboard.

1.3 Editing using input pin

In general, we can edit the value of an input pin by right clicking on it. We can also use right click drag . IOBox is a very special type of node where the contents can effect the input pin. No other node can affect its input pin.

1.4 When connected

When an input is connected, then you cannot edit that input. Since altering the value of an IOBox would alter the value of an input, you cannot interact directly with the value of an IOBox when an input is attached.

1.5 Spreaded IOBoxes

An **IOBox** can carry more than one Value. VVVV has a special way of dealing with several values at the same time, this is called a Spread.

Here we have an **IOBox** which works with 4 values. This is sometimes called a '4D vector' **IOBox**

1.6 Shortcut to create IOBox

Since you'll be making these all the time, VVVV sensibly provides you with a shortcut to create a new **IOBox**

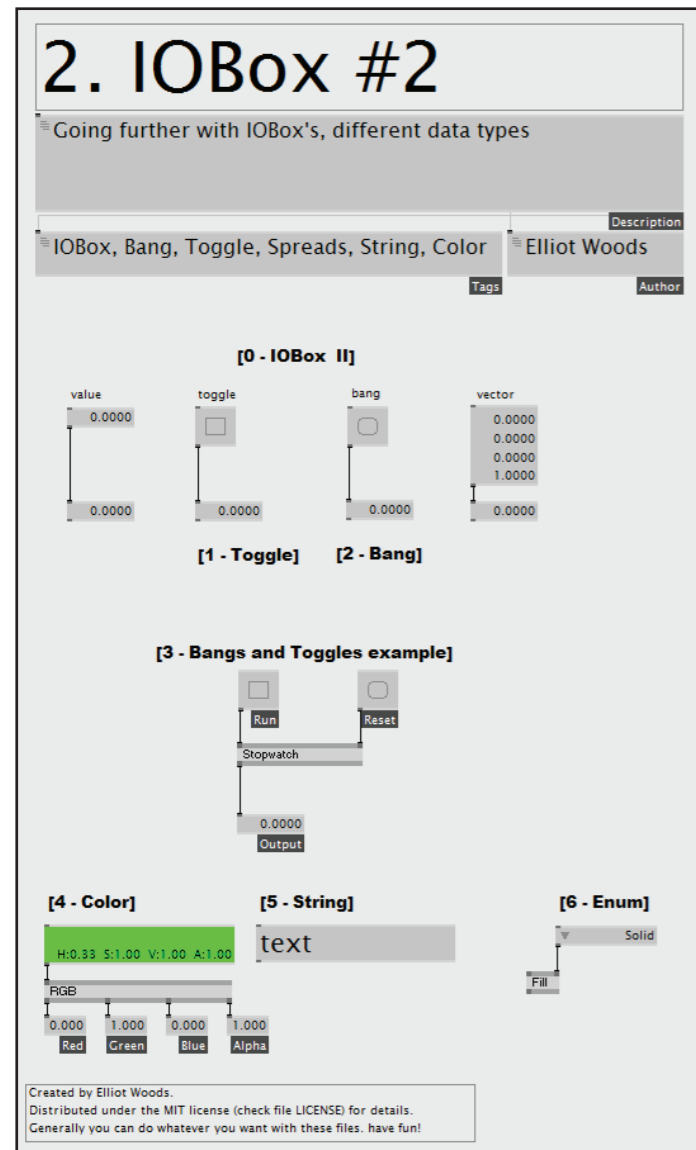
To do this double right click  in an empty area of the patch.

This will give you a simple **IOBox (Value advanced)**


A menu will also appear allowing you to create different types of **IOBox**

2. IOBox II

2.0 IOBox II



Open the example patch for this chapter. The [IOBox \(Value Advanced\)](#) supports many different ways of dealing with values.

Try interacting with these IOBox's. Remember to interact you use the right mouse button .

2.1 Toggle

A [Toggle](#) is a very simple message that can be sent around the patch. It generally switches something on or off downstream like 'enable this', 'hold this value' or 'pause this'.

VVVV thinks of a toggle as a Value of either 0 or 1. 0 denotes low or 'off', 1 denotes high or 'on'.

2.2 Bang

A [Bang](#) is another very simple message that can be sent around the patch. It generally tells something downstream to 'do something', like 'emit particles' or 'shutdown computer'.

The message only exists for an instance, then disappears. It's like poking someone on the shoulder. A bang is 'instantaneous'.

WARNING!

If you're familiar with bangs from Max/MSP, then you're likely **NOT** going to be very familiar with

bangs in VVVV. Watch out! They work very differently in VVVV!

A bang is also represented by a Value. The substance of that Value is either 0 or 1 for a bang. The Value is 1 to denote a bang being sent. A bang lasts for 1 frame, after which the value returns back down to 0.


2.3 Bangs and Toggles example




Let's try out [Bangs](#) and [Toggles](#)! These guys are the best of friends.

Open a new patch with **Ctrl** + **P** or use  to open the Main Menu and select 'new patch'


Let's create: (from top to bottom)

- 1 IOBox Toggle + 1 IOBox Bang*
- 1 [Stopwatch \(Animation\)](#)
- 1 IOBox value (hold on a sec, just create the first 2 rows)

* To create a toggle IOBox, double right click  and select 'Toggle' from the menu. Do the same for bang, but this time select 'Bang' of course.

To create the bottom IOBox, let's use another shortcut. First  on the [Output](#) pin of [Stopwatch](#). Move your mouse away and then click the middle mouse button , this should automatically create a new connected [IOBox](#) for you of the correct kind.  on the IOBox again to give it a label.

Now connect everything up. Connect the toggle you created to the 'Run' input and connect the bang to the 'Reset' pin..

We can also give names to IOBox's very easily. Since they are now connected to the Stopwatch node's [Run](#), [Reset](#) and [Output](#) pins respectively, we can copy these names into the IOBox's. To do this, we middle click  on the 3 IOBox's one by one.

Now try to interact with the 'Run' and 'Reset' boxes that you have created using right click .

Right clicking on a Toggle switches the value. Right clicking on a Bang sends a bang (switches the value high for 1 frame).

2.4 Color

A [Color](#) is a datatype within VVVV.

Colours have obvious graphical uses such as determining the colour of objects that you want to render.

There are many nodes that allow you deal with colours directly, or convert them to other data types.

2.5 String

A [String](#) is another datatype within VVVV. It allows you to deal with text.

VVVV works with UTF-8 encoded strings (international text) as well as ASCII strings (simple latin english text) + multi-line text.

Strings can also be used for dealing with URLs, Filenames, Directory names and other text-based assets. In these cases VVVV has some helpful ways of dealing with strings to make life more lovely.

2.6 Enum

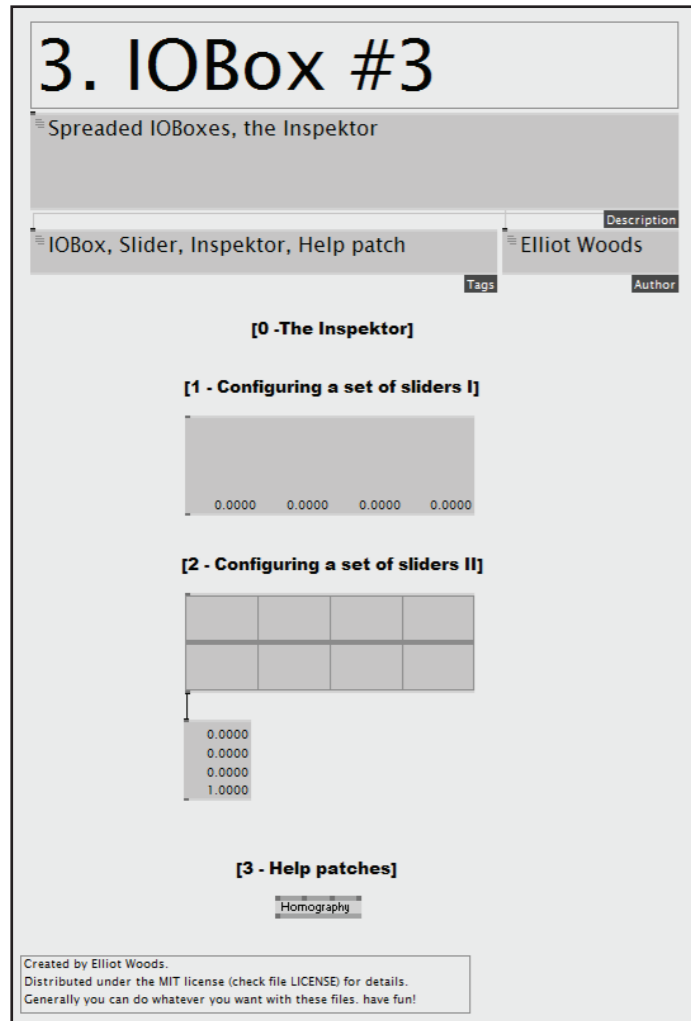
An [Enum](#) is a peculiar datatype in VVVV. It is in general the only data type to learn something from downstream. Almost all other types only send data downstream, and do not feed back any data upstream (this is quite fundamental to how VVVV works).


An enum is an enumeration of options. The node which an enum is connected to will tell it which options are available. An example of a node with an enum input is [Fill \(EX9.Renderstate\)](#).

Create this node, and try and change the right hand Input pin. You will be given a list of options for fill style (e.g. Wireframe, Solid, Point).

3. IOBox III

3.0 The Inspektor





Open a new blank patch.
 Let's create a set of sliders.
 Create a new **IOBox** with .

There are 2 pins on the IOBox at present:

- Y Input Value
- Y Output Value

However, there are many hidden pins on the IOBox that can configure it in all sorts of ways. To see these hidden pins, we need to use the **Inspektor**.

To open the Inspektor:

1. Select the **IOBox** with a single  (it should now be highlighted. To deselect an object,  in an empty area of the patch).
2. Press **Ctrl** + **I**. This should bring up the **Inspektor** window.

The **Inspektor** will show you hidden properties about whatever you have selected.


The pins shown in the **Inspektor** are split into 3 areas:

- **Config pins** - These pins can only be edited using the Inspektor.
- **Input pins**
- **Output pins**

3.1 Configuring a set of sliders I

With the **Inspektor** open and the **IOBox** selected, let's change some settings of the **IOBox**.

First let's make our **IOBox** work with a spread of 8 values at once. We do this by setting the following configuration properties:

- Set SliceCount Mode to 'ColsRowsPages' (use  to change variables)
- Set Columns to '4'

Now let's resize the **IOBox**. To do this move your cursor to the bottom edge of the node on the right hand side, your cursor should change to a resize icon. Now drag out the size of the node.


We should have 4 values shown.

3.2 Configuring a set of sliders II

Now let's change the visual style.


- Turn **on** Show Grid
- Turn **off** Show Value
- Turn **on** Show Slider
- Set Slider Behaviour to **Slider**

Let's visualise the output of this.  and select **4D Vector**. Put this beneath your sliders and connect the Y Value Output of your sliders to the Y Value Input of your 4D Vector **IOBox**.

Now try interacting with your sliders by dragging the right mouse button  up and down on each slider.

We find that the sliders can move between -1000 and 1000. We can change this range using the Minimum and Maximum Values in the **Inspektor**.

3.3 Help patches

VVVV has a built in help mechanism for most common nodes. To open a Help Patch, select a node  (e.g. an **IOBox**) and press **F1**.

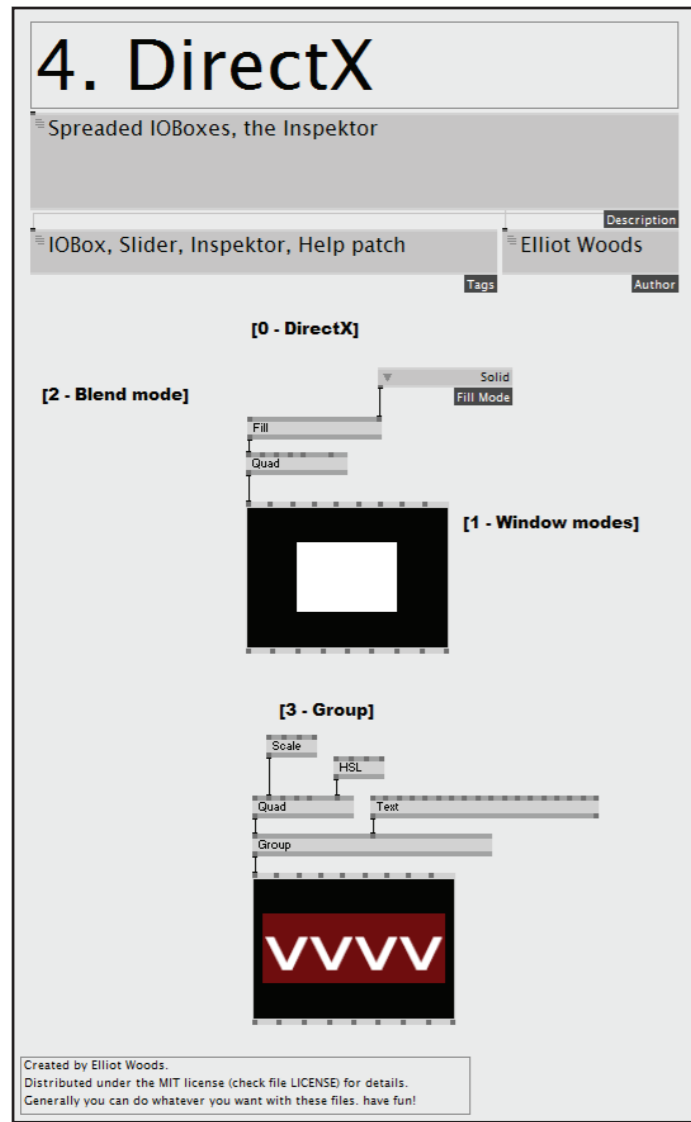
The **IOBox** help patch displays a thorough set of information about how that node works.

Let's try another one. First create a **Homography (Transform 2d)** node. Then select it  and press **F1**.

Try interacting with the top right **IOBox** using .

4. DirectX

4.0 DirectX



Now that we have a basic understanding of the VVVV interface, we can start to understand how to generate graphics.

First lets try a simple type of drawing. Let's create a **Quad** and a **Renderer (EX9)**. Connect them together from **Quad**'s **Layer** output to **Renderer**'s **Layers** input.

4.1 Window modes

With the **Renderer** selected. Try the following key combinations:

- **Alt** + **3** - Hidden
- **Alt** + **2** - Inside the patch
- **Alt** + **1** - In a seperate window
- **Alt** + **Enter** - Fullscreen

You can now press **Alt** + **Enter** again to exit fullscreen.

4.3 Group

The **Renderer** node has only 1 **Layers** input, and in VVVV, each **input** can only only accept 1 Link. So how do we connect multiple objects to 1 **Renderer**?

For this, we use the **Group (EX9)**. Let's create some nodes, from the top let's create:

- A **Quad** and a **Text (EX9)**
- A **Group (EX9)** node
- A **Renderer (EX9)**

First connect the **Group** to the **Renderer**, then connect the **Text** to the **Group**'s **Layer 2** input pin. Now you should be rendering the text "vvvv". You can change this text using the 3rd input pin **Text** on the **Text** node.

Now connect the **Quad** to the **Layer 1** input of **Group**. Both objects should now be drawing.

Since they are both the same colour, it is a little confusing to see. So let's change the colour of the **Quad**:

1. Create a new node at the top called **HSL (Color Join)**.
2. Attach the **Output** of **HSL** to the **Color** input of **Quad**
3. Change the **Lightness** pin of **HSL** to around **0.2**
4. Change the **Hue** to **0**

You should now have some white text drawn on top of a red quad.

The order in which inputs are made to the **Group** node depicts what order the objects are drawn to the screen, i.e. since the **Quad** is connected to **Layer 1**, we draw this object first, and then the **Text** which is connected to **Layer 2**. This means that the **Text** is drawn on top of the **Quad**, since it is drawn second, and because there is no other depth queues in the scene.

To change the size of the quad to fit the text, let's make a new node above **Quad** called **Scale (Transform)**. Connect the **Transform Out** of **Scale** to the **Transform** input of **Quad**. Now try editing the **x**, **y** and **z** inputs of **Scale**

4.2 Blend mode

Let's create a new node above **Quad** called **Fill (EX9.RenderState)**

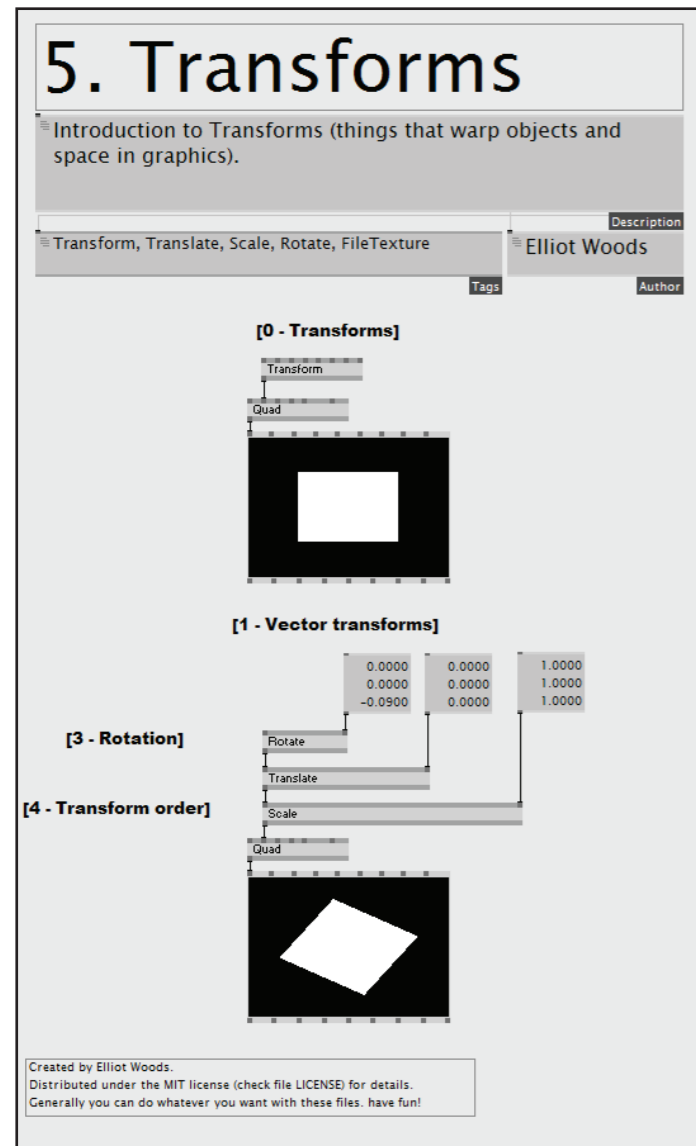
Connect it's output to **Quad**'s **Render State** input.

Now create an enum IOBox above **Fill**. This time we'll use another shortcut. First **⌘** on the top right pin **Fill Mode** to start creating a link. Then move your mouse away and do a **⌘** to create a new matching IOBox.

Try selecting different fill modes in the IOBox (remember to use **⌘**).

5. Transforms

5.0 Transforms



Let's setup a **Quad** with a **Renderer**.

Now let's add a **Transform 2d** above the **Quad**.

If we try and manipulate the input pins on the **Transform**, then we can see how we can move, scale and rotate the object in a simple way

5.1 Vector transforms

The **Transform (2d)** is good to get started and for a range of tasks, but let's also have a look at a different way of dealing with transforms which can be more elegant.

Select your **Renderer** and **Quad** by dragging a selection box around the 2 nodes. Then duplicate these 2 nodes using **Ctrl** + **D** to duplicate. You should now have a copy of the **Quad** and the **Renderer**. Move these further down the patch.

Now lets add a **Scale (Transform Vector)** and a **Translate (Transform Vector)**. Connect the **Transform Out** of **Translate** into the **Transform In** of **Scale** and connect the **Transform Out** of **Scale** to the **Transform** input of **Quad**.

Now create 2 **Vector 3D IOBox**'s using the double right click quick menu.

Connect these 2 new **IOBox**'s to the **Scale** and **Transform** node's **XYZ** inputs.

Since your scale's input value is set to **0,0,0** your quad has no size. Increase this to **1,1,1**.

Note : A quick way of doing this is to click on the **IOBox**'s **Input Value Y** and type **1** then **Enter**. This will set all slices in the spread to 1.

5.2 Rotation

Now let's add another transform to the **Transform stack**. Let's add a **Rotate** at the top in the same way we added the **Translate** and **Scale**.

The units of rotation in VVVV are :

- **0** = no rotation
- **0.25** = 90 degrees (quarter rotation)
- **0.5** = 180 degrees (half rotation)
- **1** = 360 degrees (full rotation), back to the beginning

This normalisation of rotations may seem strange, but it becomes very useful for quickly doing maths with rotations and imagining them yourself.

In VVVV, simple rotations are about 3 axes (corresponding the input **XYZ** on **Rotate (Transform Vector)**). These are:

- X - Rotation about the positive X axis (the axis pointing right)
- Y - Rotation about the positive Y axis (the axis pointing up)
- Z - Rotation about the positive Z axis (the axis pointing out of the screen towards you)

For standard 2D rotations, you'll want to spin the object in the axis pointing out of the screen.

5.3 Transform order

In computer graphics. The order in which transforms are performed is important, e.g.:

- **Rotate then translate** - Spin the object around the origin (the center of the object), then move it
- **Translate then rotate** - Move the object (hence move the center of the object), then spin the object around the origin (This causes the object to orbit around the center of the space).

Try to put the transforms in different orders to understand how this works. More on this later.

6. Spreads

6.0 Spreads

Spreads are a fundamental tool in VVVV. They are the gift that keeps on giving (metaphorically, and quite literally).

Note : For programmers: It's similar to an **Array** in normal programming speak, but also very different.

- A spread is a set of data.
- An individual item within a Spread is called a Slice
- The number of Slices in a Spread is called the Spread Count or the Slice Count
- Each Slice has a position within the Spread called a Slice Index
- The first Slice in the Spread has Slice Index of 0.
- If you try to access a slice index outside of the range (e.g. \geq spread count or < 0) then the spread 'loops'.
- A Spread of Spread Count 0 is a special case called an Empty Spread

An example

A fish and chip shop menu has 10 items, each with a price and a description. The first item on the list is "Battered cod" with a price of 2pounds 50pence. The last item on the list is "Kimchi" at 4 pounds. Therefore we could say the following:

- There are a spread of Values (price) and a spread of Strings (description)
- Both spreads have slice count of 10.
- At slice index 0 in the Value spread we have 2.5
- At slice index 0 in the spread of Strings we have Battered cod
- At slice index 9 in the spread of Strings we have Kimchi
- At slice index 10 in the spread of Strings we have Battered cod (we've looped here back to the beginning of the spread)
- At slice index -1 in the spread of Strings we have Kimchi (we've looped here back to the end of the spread)

6.1 GetSlice

A GetSlice (Spreads) node picks 1 or more slices out of a spread.

Let's create a GetSlice (Spreads) now, and attach a **4D Vector IOBox** to its Input and a simple IOBox to its Output. on the bottom IOBox to give it a name.

For readability. Let's go into the Inspektor and change some properties for both IOBox's. Set Value Type to **Integer** and Size to **14**. Also turn on Show Grid and Show SliceIndex.

Also let's add an IOBox on the Index input of GetSlice using our middle button trick . (first click on the pin to start a connection and then to make an IOBox from the connection).

Add a 1 (Spreads) to the top and connect the output to the input of the 4D vector. Set the input pin To on 1 to **4**. This node now outputs a spread of **0, 1, 2, 3** i.e. values From 0 To 4.

Now try changing the **Index** to see which slice is selected by the GetSlice node.

6.2 Always repeating

So we've seen how the data in a spread is repeated to satisfy the slice which is being asked for. This is even the case when we have 1 value.

We can test this with a GetSlice with a simple IOBox (Value) input.

When we run it through a GetSlice and roll the Index pin different slices, we get the same value.




Important note : Every Value, String, Colour, Enum, etc in VVVV is stored as a Spread, and therefore repeats like a Spread.

6.3 A visual spread experiment

So let's try and visualise what happens with spreads.

We're going to setup a simple graphical patch to use in a number of experiments. This patch will help us to visualise some spread mechanics, but also introduces a few graphical concepts as well.

Follow these steps:

1. First create a **Quad** and a **Rendered** and connect them up. You should get the familiar white square in a black box.
2. Now lets make a new node called **FileTexture** above **Quad**. Connect the **Texture Out** output of **FileTexture** to the **Texture** input of **Quad**.
The **FileTexture** node loads in an image file, and creates a **Texture** which you can apply to your graphical objects.
Textures are stored on the graphics card, and can be used in all sorts of ways to manipulate your objects with image data (e.g. image maps, height maps, bump maps, etc).
3.  on the **Filename** input of **FileTexture** and select an image from your own computer. You should now see that image applied to the quad.
4. Now let's load a specific texture. We go to the VVVV folder where you have it installed. We go to the 'girlpower' folder. Then we go to the 'images' folder (vvvv/girlpower/images). Now select the file "**ring thin.bmp**". We should now be drawing a circle to the screen. If the **Rendered** is square, the circle will be square. But if the **Rendered** is squashed, so will the circle be.
5. To straighten the circle, we need to counteract the **Aspect Ratio** of the **Rendered**. We do this with some simple maths. Create a **/ (Value)** node (this node divides numbers. i.e. $4/2 = 2$).
Attach the 2 inputs of the **/** to the **Width** and **Height** output pins of the **Rendered** respectively.
Create an **IOBox** and attach it to the **IOBox**'s output to visualise the output. This number represents the aspect ratio. Go into **Inspektor** and set the **Descriptive Name** of the IOBox to **Aspect Ratio**.
Now add a **Scale (Transform)** (not vector version!) above your **Quad**. Connect to your **Quad**'s **Transform** pin. Connect the output of your **Aspect Ratio** to the **Y** input of **Scale**. The aspect ratio should now be fixed for your quad, regardless of the **Rendered**'s aspect ratio.
Because you now have a link going upwards in the graph, it looks a little messy. Feel free to select it and tidy it up using either **Ctrl+Y** to curve it, and/or **Ctrl+H** to 'hide' it.
6. Scale down the quad using a **UniformScale (Transform)** node. Attach this to the input of your **Scale** node and reduce the **XYZ** input to **0.1**.
Remember you can edit the value both by on  the pin and then typing a or  on the pin and move the mouse up/down.
7. Create a **Translate (Transform)** node (again, not the vector version!). and attach that to the **UniformScale**'s input

Ok before going any further. Take some time to figure out what's happening. This are some very common VVVV fundamentals here which are worth understanding (Transforms, Layer, Texture, Maths).

Notice how everthing is wired up for 1 object. There's 1 aspect ratio correcting scale, 1 uniform scale, 1 texture, 1 quad.

Now let's step it up! Create a **LinearSpread** node above the **Translate** and attach the **Output** to **Translate**'s **X** input. Now check the top right input pin of **LinearSpread**, **Spread Count**. Let's increase this value.

As you see you get lots of circles rendered to the screen. Check the **Output** of **LinearSpread** by rolling your mouse over the output (you dont need to click). It might look something like this:



Output: (3) -0.3333~

This means there are 3 slices in the spread, and the first slice has a Value of approximately -0.3333 (the ~ denotes approximate).

6.4 - Olympics 2012?

Ok. Let's make a copy.

1. Select that section of the patch we just made.
2. Press **Ctrl** + **D** to duplicate. Dont press anything else.
3. The bits you've duplicated are all mixed up with the old bits, but your new bits are all selected.
4. Use the **Down** key on your keyboard to move the nodes down within the patch.
To do this quicker, use **Shift** + **Down**. Do this until you've moved the new nodes into a clear section of the patch

Let's get rid of the background. Delete **RetroColors** by clicking  on it and then pressing **Delete**.
Right click  on the **Background Color** pin to reset it to **black**.

Let's make the rings bigger. Set the **UniformSpread** to **0.4**. This is a quick way of saying 'set the parameter of UniformSpread (in this case **XYZ**) to value **0.4**'.

Create a new **LinearSpread** and connect it to **Translate**'s **Y** input. Set the **Spread Count** of the 2 **LinearSpread**'s to **5** and **2** respectively.

Adjust the **Width** pins of the **LinearSpread**'s until you get the olympic rings.

For good measure, add some colour to the rings by either using a **RetroColors** with an **I (Spreads)** on the **IndexHSL** (Color Join) with a **LinearSpread** on the **Hue** input pin. Also set the **Draw Mode** of the **Blend** node to **ColorAsAlphaBlend**.

6.5 Tidying the blend mode

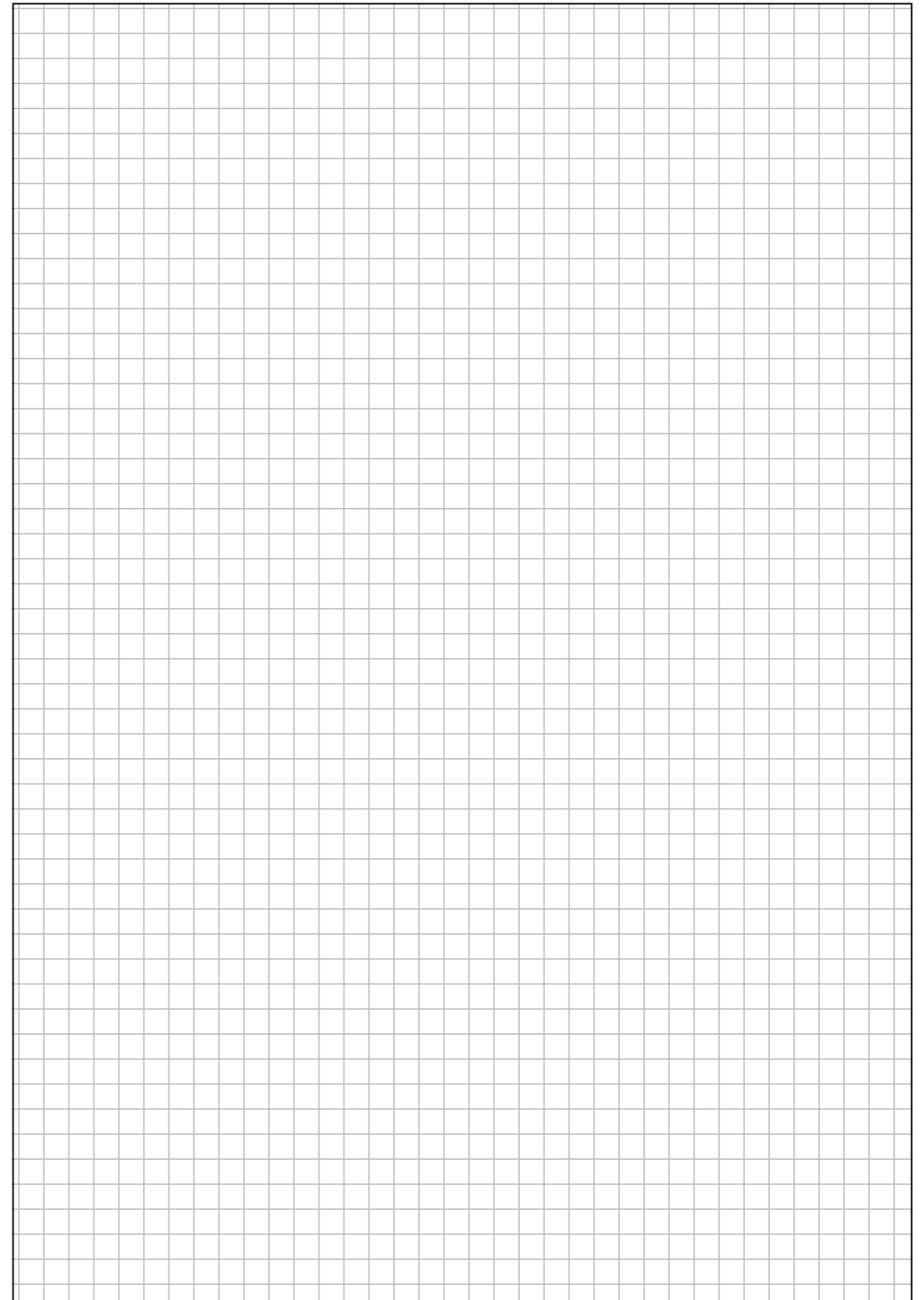
If we have too many circles, they will start to overlap. Since the textures have an opaque black background, they will interfere with each other. We can see that the circles are opaque by changing the background colour of the `Renderer`.

To change the background colour of the `Renderer`, create a `RetroColors (Color)` and connect its `Output` to the `Background Color` input pin on the `Renderer`. Select different colours by changing the `Index` input of `RetroColors` and/or changing to a different games console colourset using the `Mode` pin.

On the right hand input of `Blend` we see < That was fun, but anyway, we can definitely see that the background of the quads are black. Let's change the `Blend Mode` of the quads, (Photoshop users will be familiar with this idea).

Create a `Blend (EX9.RenderState)` above `Quad` and attach the output of `Blend` to the `Render State` input of `Quad`. Set the `Draw Mode` of `Blend` to **Add**.

Now when the quads are drawn, their colour is 'added' to whatever is behind them, i.e. since black is **0** and **a + 0 = a**, black has no effect when the `Draw Mode` is set to **Add**.



7. Animation

7.0 Introduction to animation

The screenshot shows a VVVV patch titled "7. Animation" with the subtitle "Making things move using Animation nodes." The patch is organized into several sections:

- [0 - Introduction to animation]**: Contains two sub-sections:
 - [1 - Damper]**: A patch with a Switch node, a Damper node, and a value of 0.000.
 - [2 - Decay]**: A patch with a Switch node, a Decay node, and a value of 0.000.
- [3 - LFO driven animation]**: A patch showing an LFO node connected to a LinearSpread node, which is then connected to a Translate node. The LFO node has a value of 0.1756 and a Cycles output of 36.
- [3 - RandomSpread animations]**: A patch showing a Blend node connected to a LinearSpread node, which is then connected to a Translate node. The LinearSpread node is also connected to a RandomSpread node.
- [3 - Timeliner]**: A patch showing a Timeliner node.

At the bottom left, there is a small text box: "Created by Elliot Woods. Distributed under the MIT license (check file LICENSE) for details. Generally you can do whatever you want with these files. have fun!"

There are an infinite number of ways to animate things in VVVV. Here are some obvious ones:

- Damping between 2 states
- Using **Timeliner**
- Using LFO

7.1 Damper

A simple form of animation is ‘damping’ between 2 states. Examples of nodes which can cause this type of animation are:

- **Damper**
- **Decay**
- **DeNiro** (so called because it acts like a ‘taxi driver’)
- **Newton**

These nodes tend to take an Input which changes, and give an Output which moves smoothly. The speed and type of this movement depends on the type of the node, and the parameters of that node.

Let’s try the following:

1. Create a **Damper (Animation)**.
2. Create 2 **IOBox**’s around the **Damper**. One attached to its Input and one attached to its Output
3. Create a **Toggle IOBox** using the **•|** shortcut menu, and attach its output to the input of the top **IOBox**

Now try toggling the **IOBox** on and off. You will notice the Input jumps sharply between values, whilst the output moves smoothly.

7.2 Decay

Try this again, but with a **Decay** node instead of a Damper node.

Set the Decay input pin of **Decay** to **1 second**. Now when you make the value high it jumps instantly. But when you make it low again it takes 1 second to return back to **0**.

7.3 LFO driven animation

Firstly let’s grab the rendering patch we had from last time. Copy it **Ctrl** + **C** there, and paste it **Ctrl** + **V** here. Set the Spread Count to **3**.

To save some space. I’ve deleted the **Aspect Ratio IOBox** and connected the **/** directly to the **Scale**’s Y input. Also I’ve scaled down the size of the **Renderer** in my patch.

First let’s manually ‘roll’ the Phase input of the **LinearSpread**. Just put your mouse over **•|** it hold the right mouse down and drag up/down.

Now create a new node **LFO (Animation)** and add 2 **IOBox**’s to the Output and Cycles output pins on it. Set the Period input on **LFO** to **3 seconds**. Notice how the Output and Cycles are affected.

Now connect the Output of **LFO** to the Phase input of **LinearSpread**. You should see the circles marching in a line.

7.4 RandomSpread animations

Now make a copy of that section of the patch, delete the **LFO** with its **IOBox**'s. The **Phase** of the **LinearSpread** will now keep the last value sent to it.

Create a **RandomSpread (Spreads)** node above **Translate**, and attach the **Output** of **RandomSpread** to the **Y** input of **Translate** and set the **Spread Count** to **3**.

RandomSpread is now outputting a Spread of 3 values, each of which should be random (if you want to get nerdy, pseudo-random).

If we roll the value of the **Random Seed** input pin of **RandomSpread**, we'll get a different set of random numbers.

Create a new **LFO** and connect the **Cycles** output to the **Random Seed** input. Now periodically the circles will jump to new positions.

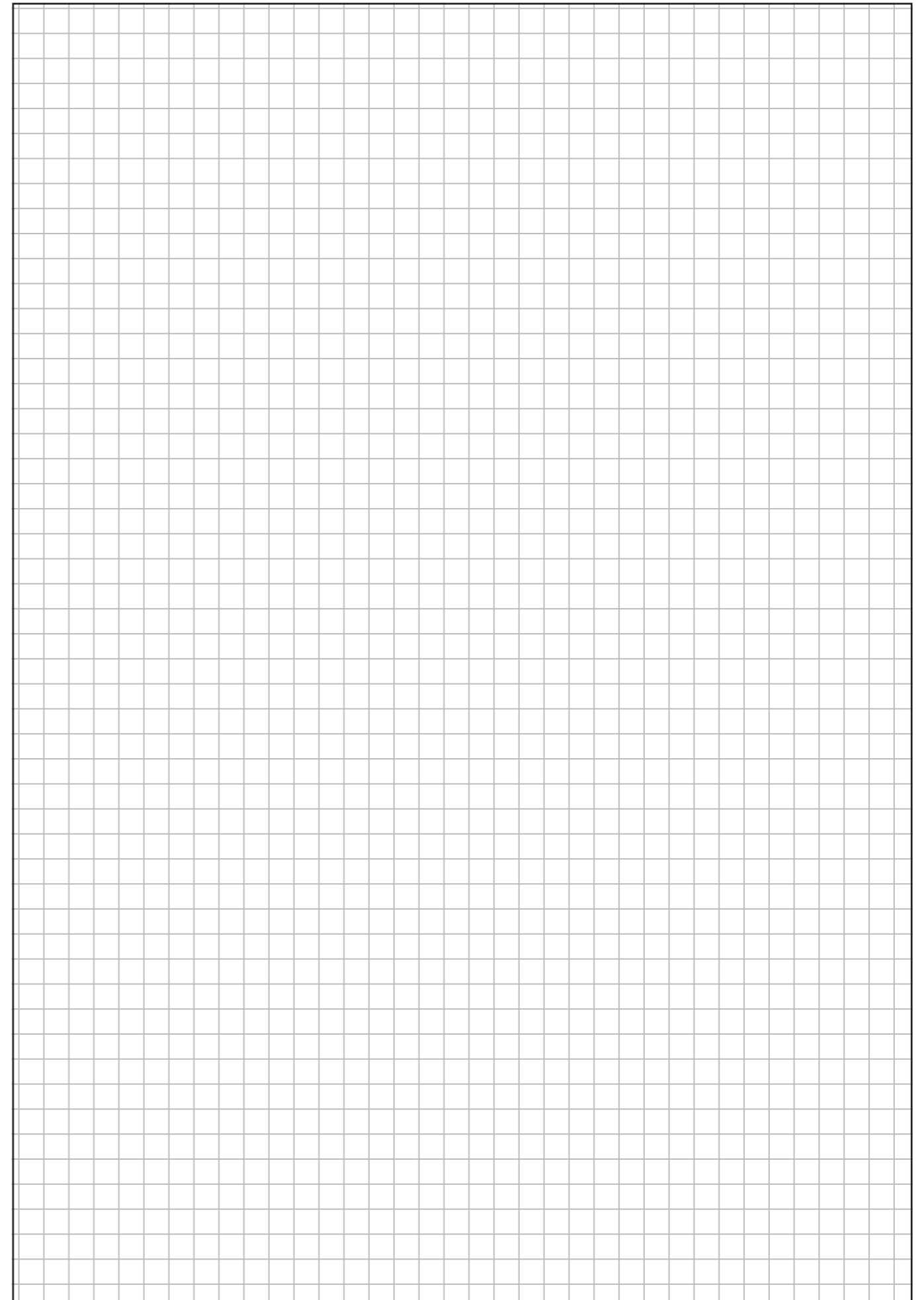
Aha! We know how to make things move smoothly instead of jumping dont we? Let's stick a **Damper** in the path between **RandomSpread** and **Translate**. Now they should move smoothly!

To improve this, let's try a few things:

- Change the **Width** of the **RandomSpread** to **2**. This means that the range of values output will be between **-1** and **+1** which is the range of Y values that fit inside the Renderer. The range is defined by the **Width** and the **Input** (i.e. the center) of **RandomSpread**
- Try increasing the **Filter Time** of **Damper** to make the circles accelerate more slowly.
- Try changing the **Period** of the **LFO**. Note that if the **Period** is too short (i.e. quick), then the input to the **Damper** might move too quickly for the output to ever catch up.
- Try increasing the **Spread Count** on **RandomSpread**
- Try changing the **Spread Count** and **Width** of the **LinearSpread**

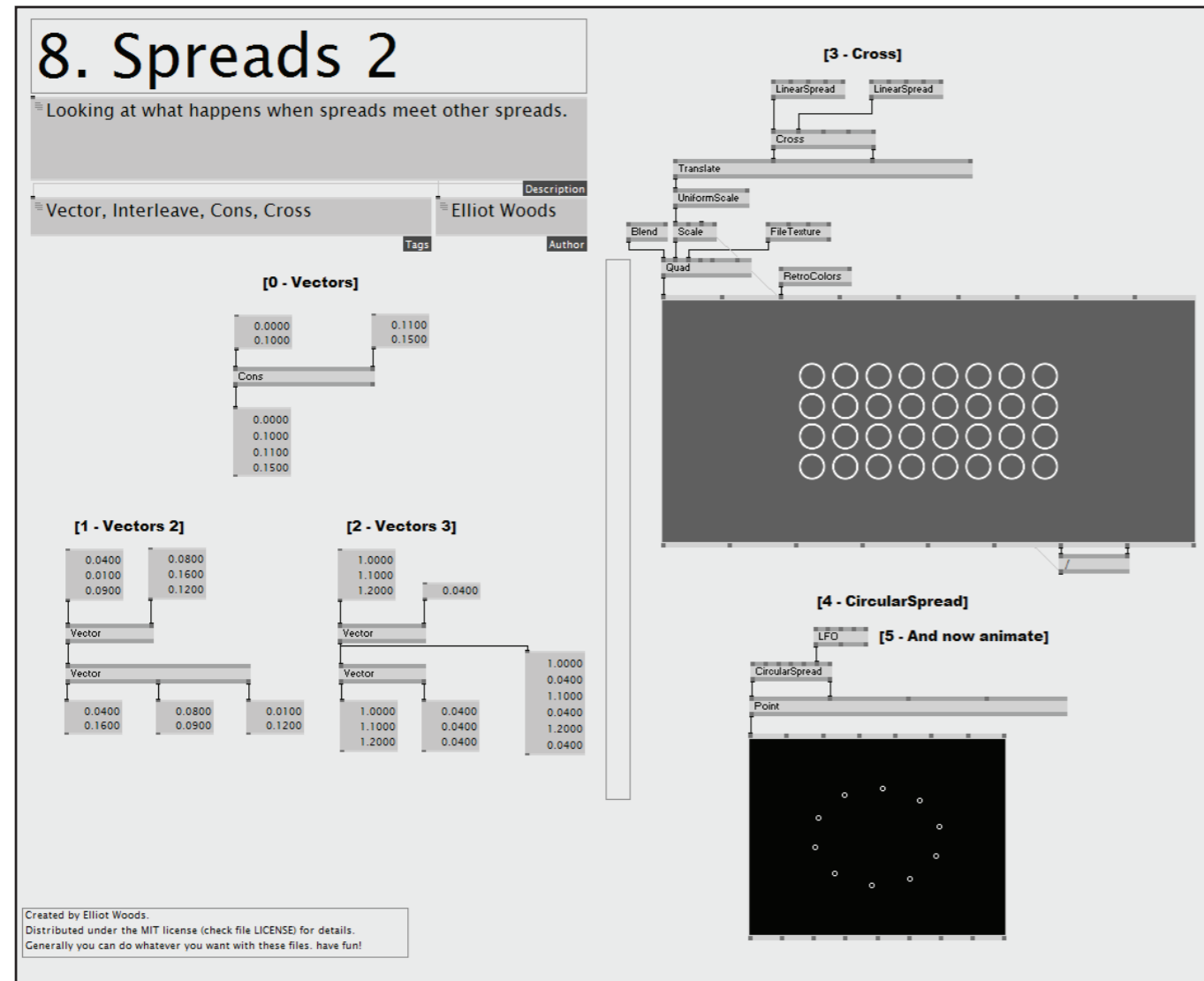
7.5 Timeliner

The **Timeliner** node is a fantastic tool for advanced animations. Check out the help patch for it by creating the node and pressing **F1** whilst it is selected.



8. Spreads II

8.0 Vectors



A Vector is represented by a Spread. A Spread of Vectors is represented also by a Spread.

A Spread with Spread Count 6 could either be a Spread of 3 2D Vectors, or a Spread of 2 3D Vectors. VVVV is totally agnostic about this. It just sees numbers.

Let's try this out:

1. Make 2 **Vector 2D** IOBox's.
2. Make a Cons (Spreads) node
3. Make a **Vector 4D** IOBox

Connect the 2 2D IOBox's to the 2 inputs of Cons. And connect the 4D IOBox to the output of Cons.

If we edit the values in the top boxes, we can see the values change in the bottom box. The 2 2D vectors become a 4D vector.

The Cons node glues spreads together. It is short for Concatenate

8.1 Vectors 2

Now try this situation:

1. 2 **3D Vector** IOBox's attached to the inputs of a Vector (2D Join)
2. Create a Vector (3D split) and connect the XY output of Vector (2D Join) to the XYZ input of Vector (3D Split).
3. Add 3 **Vector 2D** IOBox's to the 3 outputs of Vector (3D split)

Here we must embrace that the numbers are free to lose their identity at any point. We've input 2 3D vectors, which arrive at the Vector (2D Join). At this point all the values on the left become X values, and all the values on the right become Y values. Therefore the identity of the numbers become **3 sets of 2D vectors** or in VVVV speak (and specifically, what comes on the output of that node) **A Spread of 2D Vectors with Spread Count 3**.

In this case, the **Vector 3D** IOBox's simply become convenient input methods for spreads of 3 values.

Then anyway, we lose our **A Spread of 2D Vectors with Spread Count 3** identity again, by passing the spread into the Vector (3D split) node. At this point the spread of 6 values is interpreted as **A Spread of 3D Vectors with Spread Count 2**.

This is then split out onto the output IOBox's

8.2 Vectors 3

OK. Let's keep up :)

Now try:

1. 1 **3D Vector** IOBox attached to the X input of a Vector (2D Join)
2. 1 ordinary IOBox attached to the Y input of the node Vector (2D Join)
3. Create a Vector (2D split) and connect the XY output of Vector (2D Join) to the XY input of Vector (2D Split).
4. Add 2 **Vector 3D** IOBox's to the X and Y outputs of Vector (3D Split)

Here we have a spread of 3 2D vectors. And we see that the Y value is repeated. We can visualise this more by:

1. Create an IOBox
2. Goto the Inspektor Ctrl + I
3. Set the Rows of the IOBox to 6
4. Connect the XY output of Vector (2D Join) to the input of the new IOBox

Notice how the Y value interleaves the X values in the Spread.

8.3 Cross

Let's grab our rings patch from before.

Set the Spread Count of the LinearSpread to 4. Now duplicate the LinearSpread and attach the second one to the Y input of Translate

Ok we've got 2 spreads going in, each of Spread Count 4, and we've got 4 rings being rendered. But what if we want a grid of 4x4 rings? Then we use the Cross node!

Attach the 2 LinearSpread's to the X In and Y In inputs of the Cross node respectively.

Connect the X Out and Y Out output pins of Cross to the X and Y inputs of Translate.

You should now have 16 rings instead of 4.

You may also notice that it's possible to use the same LinearSpread for both inputs on the Cross as they are giving identical output.

Otherwise you can make a different grid (e.g. 6x4).

8.4 CircularSpread

Now to keep the patch simple, we will use the GDI (Renderer):

Create a Renderer (GDI) and use Alt+3 to put it into the patch (rather than a separate window).

Create a Point (GDI) and attach its output pin to the Layers input pin of Renderer

Now we should have a small cross rendered the the screen. We can change this graphic by changing the Type pin on the Point node.

Now create a CircularSpread node above the Point, and attach the Output X and Output Y pins of CircularSpread to the X and Y inputs of Points respectively.

Increase the Spread Count on the CircularSpread (e.g. to **10**).

You should now have a circle of circles.

8.5 And now animate

Add an LFO to the Phase pin of CircularSpread.

Reduce the Period of the LFO to **3 seconds**.

9. Transforms II

9.0 Object, World, View, Projection

Within the 3D world of computer graphics, there are many spaces. The most general ones are:

- Object - The coordinates within the object
- World - The coordinates with the world containing all the objects. When we transformed our quads before, we were transforming them in the world space. This is a Euclidean coordinate system, i.e. right angles are preserved
- View - The World transformed from a particular viewpoint's position (transforming a position is commonly called a translation) and rotation. This space is still Euclidean
- Projection - This is the coordinate system of the camera onto the scene. When a perspective transform is applied to the scene, then this space is non-Euclidean, and is instead Projective

Alternatively we can also think of Projection Space (especially when projection mapping) as the coordinate system of the projector. This then becomes the coordinate system which has to match up with real world objects that we are projecting onto

9.1 3D Camera

It may be difficult to admit, but much of our lives are 2D. Our eyes see fundamentally in 2D, and its only by having a pair of them that we get a sense of 3D. But it is only a sense, and in truth, we are always limited to seeing 2D projections of things. This is generally true in computer graphics, especially when displayed on a monitor.

So how do we see something in 3D? Well, we get a camera that we can move!

Create:

1. A Renderer (EX9) and put it into the patch using Alt + 2
2. An AxisAndGrid (DX9). Connect this to the Renderer
3. A Camera (Transform Softimage).

Now hook up the camera.

Connect the View output of Camera (2nd output) to the View input of Renderer. Do the same for Projection

Now we can see the axis and grid from a perspective viewpoint.


You can use the following controls to manipulate the camera:

- O + ↓ - Orbit
- P + ↓ - Dolly (fast)
- P + ↓ - Dolly (slow)
- Z + ↓ - Move
- Z + ↓ - Zoom
- Hold R to reset the view

Be careful not to destroy your patch with all those mouse actions!

9.2 Modules

Bits of code that you want to use all the time can be wrapped up in packages called Modules.

Modules are in fact simply  subpatches that you use quite often. This Camera node is in fact a module made by a VVVV user. To see inside it, on it.

Press Alt + 3 when you're done to hide the patch.

9.3 3D vector transforms

Create a copy of the patch we just made, and insert a Group (EX9) between the AxisAndGrid and your Renderer

Create a Sphere (DX9) and attach it to the Layer 2 input of Group.

Add a Translate (Transform Vector) and attach the output to Sphere's Transform input.

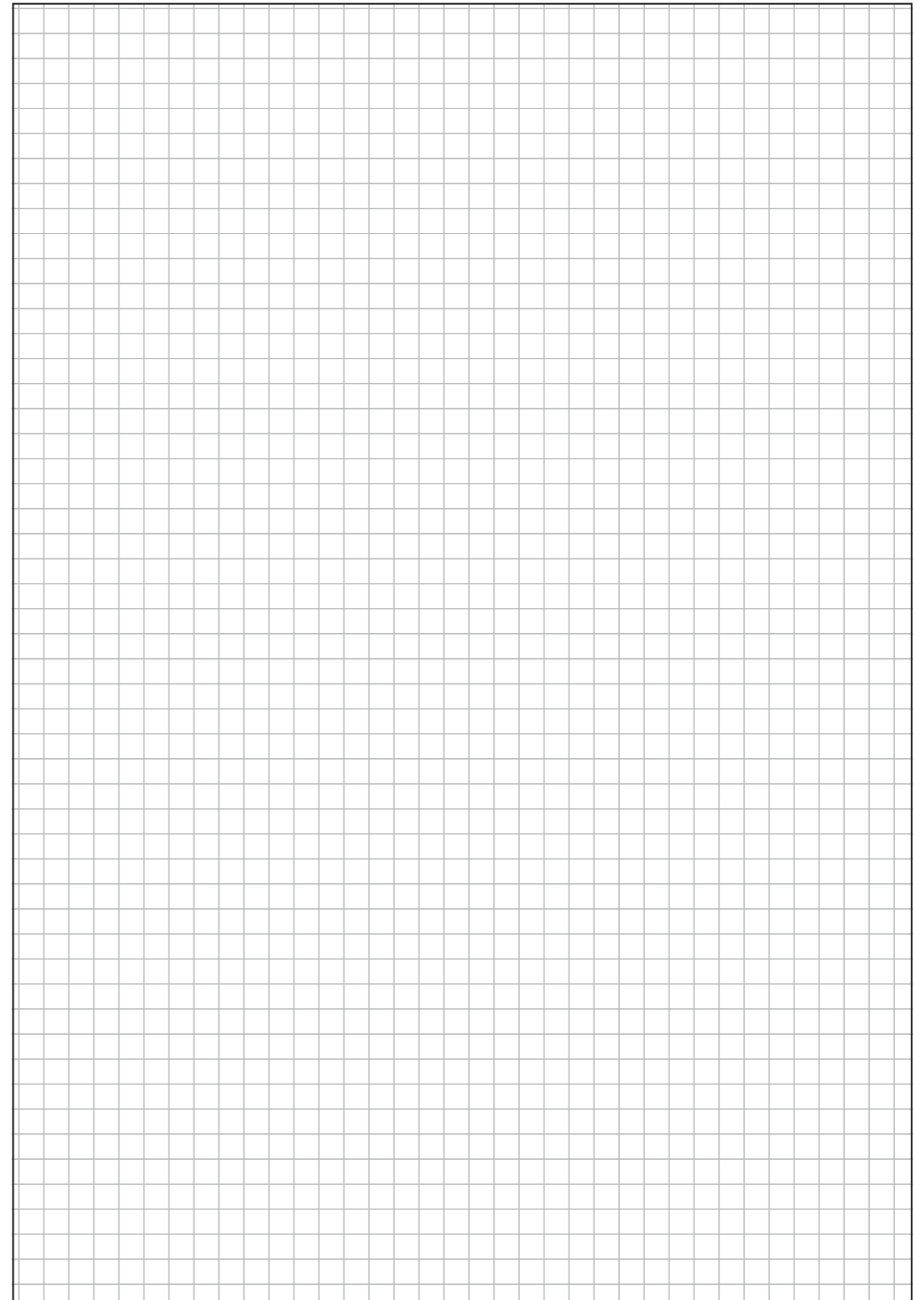
Create a RandomSpread and attach its Output to the Translate's XYZ input.

Add an IOBox to RandomSpread's Spread Count input.

Give that IOBox a value of **120**. This means you have 40 3D vectors, and therefore 40 spheres.

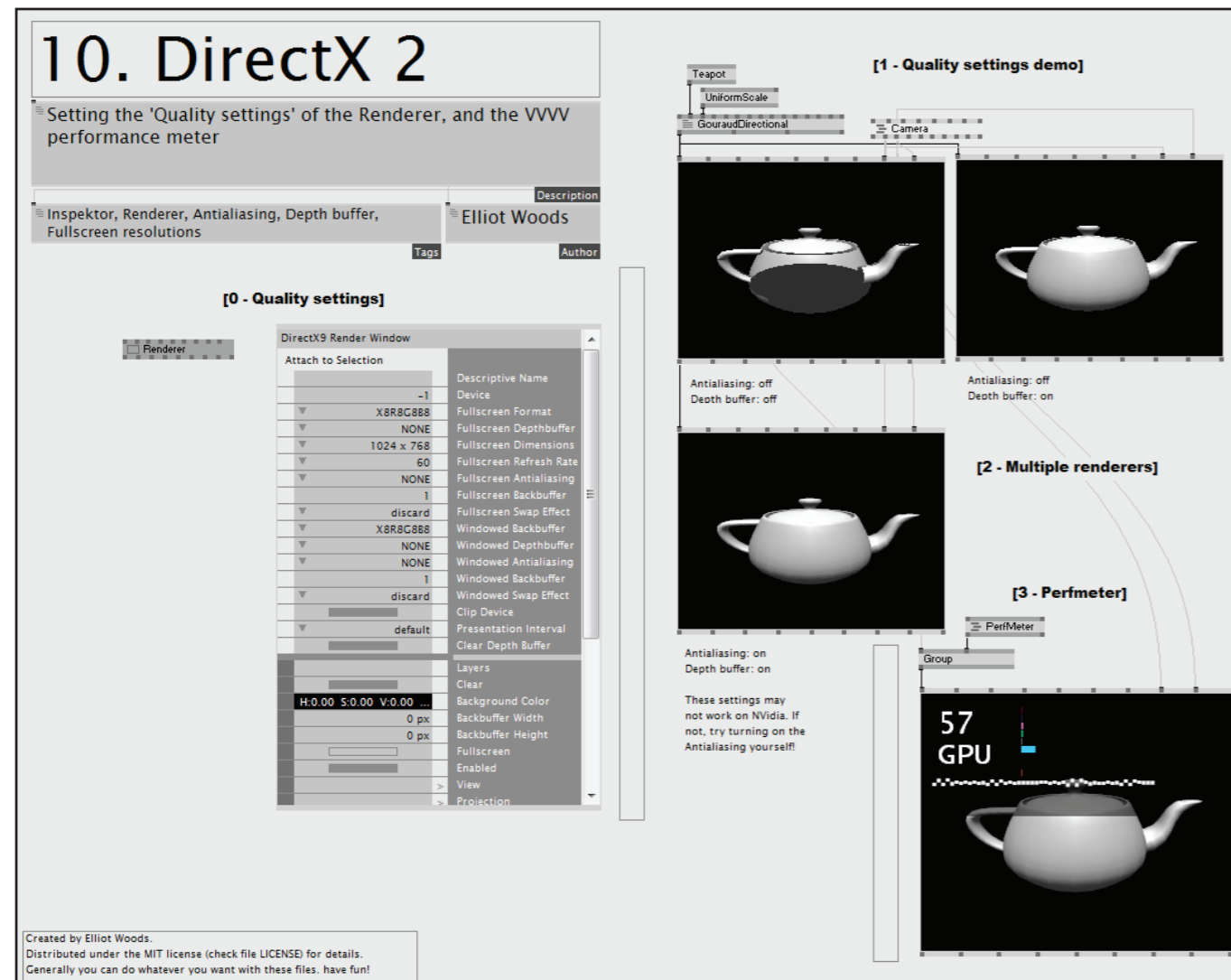
To make them look less blocky. Increase the Resolution X and Resolution Y pins on the Sphere node. Be careful not to go too high, especially on older graphics cards!

Now try to give the spheres some colour, and make them move.



10. DirectX II

10.0 Quality settings



The most important thing most of the time is to make sure to turn on Antialiasing and Depthbuffer. To demonstrate this, I've created 3 renderers at the side which share a common scene and a common camera. They each have different settings for Windowed Antialiasing and Windowed Depthbuffer.

WARNING: The Inspektor locked inside this patch will block you opening up another Inspektor. But if you turn off the Attached button, then it will start acting like a normal Inspektor (albeit inside the patch rather than in a window).

10.1 Quality settings demo

Here we demonstrate 3 Renderers with different quality settings so that you can compare and identify the differences.

Feel free to patch this out for yourself, but for the sake of this tutorial, we wont go through it step by step.

Another new bit is the GourandDirectional shader (more on that in the next tutorial!).

10.2 Multiple renderers

Here we're also demonstraing something that VVVV does effortlessly:

Rendering the same objects to multiple Renderer's.

In fact, with VVVV you can mix and match Renderer's however you like. It's a real strength of the platform. And something that isn't properly available on any other hardware accelerated platform for media arts use.

10.3 Perfometer

The PerfMeter (Debug) node gives you vital information about how your computer is performing and what is slowing it down.

The big number is the Framerate which is measured in Frames Per Second (fps). You generally want this to be the same as your Refresh rate (commonly 60Hz, therefore 60fps is best).

The running graph is the history of your framerate, and the other graphs give you more detailed information as to what is taking up your processing time.

An ideal situation is generally a nice flat line at 60fps.

The Renderer (EX9) has plenty of settings hidden away in the Inspektor. Here are some of the ones you definitely want to become familiar with:

- Fullscreen Depthbuffer - For 3D scenes you generall want to turn this 'on', i.e. choose a setting other than **NONE**. The Depth Buffer is what that graphics card uses to detect when objects are in front of each other, so it knows how to draw them properly
- Fullscreen Dimensions - When you make your renderer fullscreen, this is the resolution that it will use. The renderer will become fullscreen on whatever screen it is on at that time, i.e. if you want it to come up on the second screen, you must drag the renderer window to there before going fullscreen. Also by doing this, you will have the correct list of resolutions for that screen
- Fullscreen Antialiasing - This smooths the edges of objects, generally turn this on to make graphics look detailed and smooth. Depending on your graphics card different options will be available.
- Windowed Depthbuffer - Same as Fullscreen Depthbuffer, but this option applies instead when the renderer is not fullscreen
- Windowed Antialiasing - Same as above

11. Shaders

11.0 What is a shader?

Created by Elliot Woods.
Distributed under the MIT license (check file LICENSE) for details.
Generally you can do whatever you want with these files. have fun!

A Shader is an advanced piece of graphics programming which runs on the graphics card. It lets a developer specify in detail how an object should be rendered to the screen.

In VVVV, shaders are commonly called Effects, and are represented by a file with extension **.fx**.

The idea of modern shaders was created (afaik) by **Pixar** as part of their **RenderMan** rendering package in the late 1980's. Their shaders ran on the CPU and took a long time to calculate for a scene. With the introduction of new graphics cards (Around the time the GeForce FX was released), it became possible to write programs that ran directly on the GPU (Graphics Processing Unit).

The GPU is capable of calculating lots of small programs in parallel (sometimes hundreds of programs simultaneously). These programs are called shaders and be used to:

- Geometry shader - Generate geometry
- Tessellation shader - Increase an object's detail
- Vertex shader - Manipulate vertices
- Pixel shader or Fragment shader - Control pixel by pixel rendering

11.1 Basic shader usage

In VVVV, you apply a Effect to a Mesh. The shader may accept 1 or more Textures, and will always accept at least 1 Transform (this is the World transform which is accepted on the generically named Transform pin).

The most basic shader packaged with VVVV is called Constant (EX9.Effect). Create one of these now and attach it to a Renderer.

The Renderer will remain black, as there is no Mesh attached to the shader. There are a number of common meshes built into VVVV:

- Box
- Sphere
- Grid
- Teapot
- Cylinder
- Torus

You can list the available meshes by searching for **EX9.Geometry** in the NodeBrowser. Let's attach a Cylinder (EX9.Geometry) to the Mesh of the Constant shader node.

Because we are looking side on, we see a square. Let's add a Camera, and Group in an AxisAndGrid (DX9) to get some context.

Turn on the Antialiasing and Depth Buffer quality options.

We should see a very 'flat' looking cylinder within the scene.

11.2 Other shaders


There's lots of great shaders to make smooth results. Some examples of these are:

- GourandDirectional
- GourandPoint
- PhongDirectional
- PhongPoint

And there are many online in the Contributions area of the VVVV website.

Let's have a look at some now.

Copy the patch you just made with the earth cylinder and delete the texture transform (LFO and Translate).

Double click  on the Constant node to bring up the NodeBrowser. Select PhongPoint. You will notice that this node has many more input pins, to allow you to change many properties regarding how the shader is drawn.

Use the same method to switch the `Cylinder` mesh with a `Sphere (EX9.Geometry)`.

Create a `Translate (Transform Vector)` and attach it to the `Transform` of the shader. Attach a `RandomSpread` to the `XYZ` input of the `Translate`, and set the `SpreadCount` to **30** to create 10 3D vectors, i.e. 10 spheres. Set the `Width` of the `RandomSpread` to **5**.

Now let's animate using a `DeNiro` and `LFO` like we did before.

11.2 Texture transforms

Let's add a texture to the `Cylinder` using `FileTexture`. Select the Earth **512x512.jpg** image from within the `vvvv/girlpower/images` folder.

Now add a `Translate (Transform)` and connect it to the `Texture Transform` input pin on the `Constant` shader. This allows you to move the position of the texture on the object. Try rolling the `X` input value on `Translate`.

Let's add an `LFO` and connect the `Output` to the `X` pin of the `Translate`.


11.3 Lighting

Notice that one of the input pins of the `PhongPoint` is called `Light Position XYZ`. This pin accepts a 3D position for the light.

Create a **3D Vector** `IOBox` and connect it to this input pin. Now the light should be at the origin **(0,0,0)**.

This kind of light source is a point light source (i.e. a light at a particular 3D position within the scene). If you want the objects to react to a directional light source, then use `PhongDirectional` or `GourandDirectional` instead of the point versions.

11.4 HLSL

Shaders are written in a language called `HLSL`. To see this code,  on the `PhongPoint`. This brings up the `Code Editor`.

As you get more advanced with VVVV, you can try editing these shaders yourself.

If you want to learn more about shaders, I suggest looking at http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html (I've got that book and that's how I learnt!).

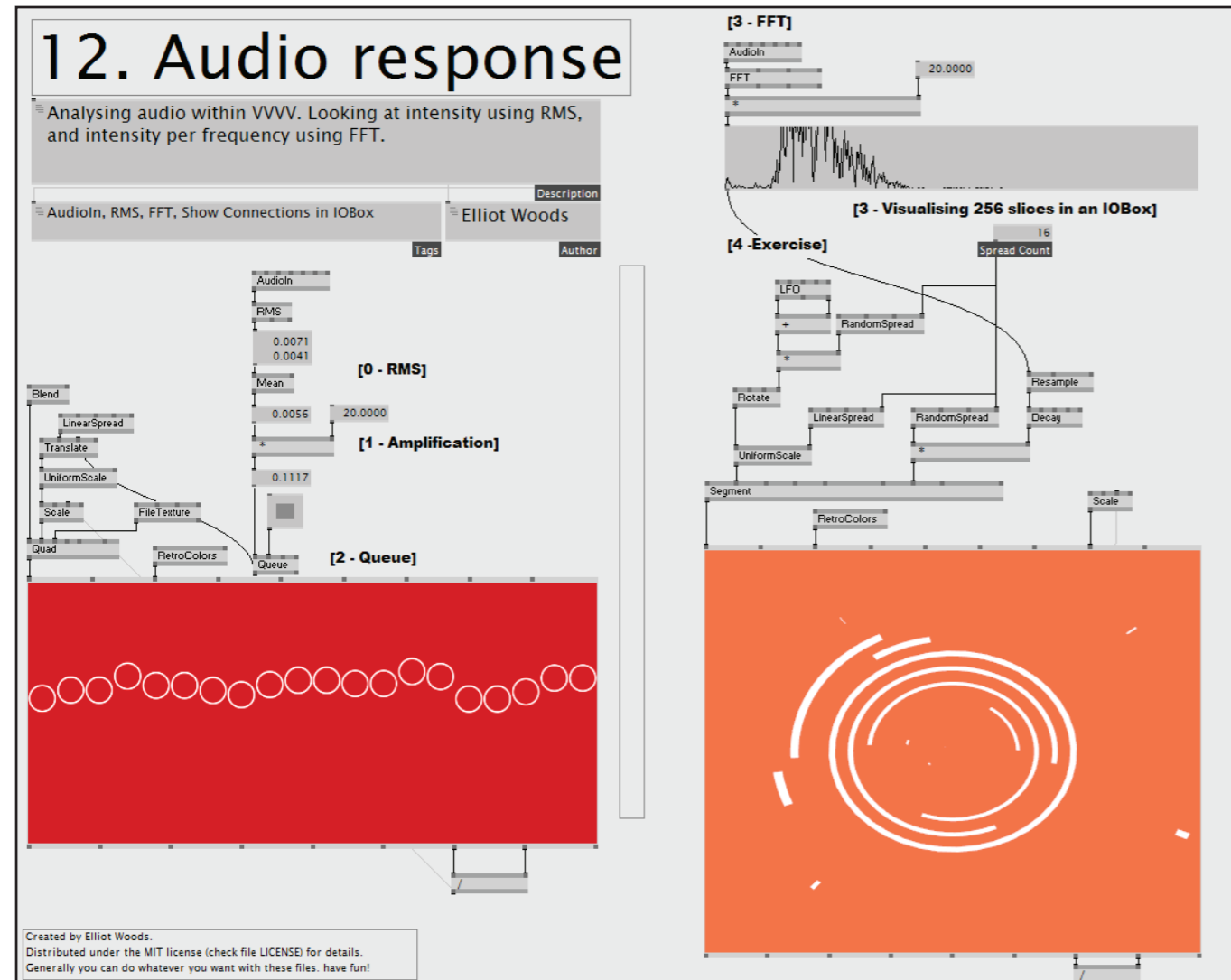
11.5 GPU calculations

GPU's have become much more powerful than CPU's (50-100x more powerful), and can therefore perform much more calculations than a CPU can within the time of 1 frame (1/60th of a second).

Many developers are therefore exploiting GPU's to perform calculations. One example of this is the **GPU Particles** library in the contributions section of the website, which allows you to manipulate and render 1000's of particles at the same time by harnessing the power of the GPU.

12. Audio response

12.0 RMS



Ok, presuming there's a microphone on the computer you're using, then we can measure the sound in the environment. This could be music, speech or whatever.

Let's create an `AudiIn` node, and connect it to an `RMS` node. Change the `Enabled` pin on `AudiIn` to `1`. Create a **2D Vector IOBox** and attach it to the output of `RMS`. The numbers should show the audio level.

The 2 numbers represent the left and right channels. Create a `Mean (Spectral)` node to average out these 2 values. Attach the output of the `IOBox` to the `Mean`'s input.

Put a normal `IOBox` on the output. Let's grab the 'Audi' patch (the one that renders rings) from our previous Spreads tutorial.

12.1 Amplification

This value is quite low, so let's feed it into a `(Value)` node and put another `IOBox` on the other input of the `*`. Set the value of the second `IOBox` to something like `20`.

Put an `IOBox` on the output of the `*`. Adjust the amplification factor (the number in the second `IOBox` until the values start varying between `0` and `1` when sound is being detected.

12.2 Queue

If we want to keep a history of these values, we can use the `Queue (Spreads)` node. Let's create one now.

Connect the sound intensity to the `Queue`'s `Input`. Set the `Frame Count` to `100`. Create a **Toggle IOBox** and set it to high (i.e. `1`) and connect the output to `Insert` on the `Queue`. Now it will keep a history of the last 20 sound intensity values. These are output in a spread of `Spread Count 20` on the bottom of `Queue`.

Copy in the 'Audi' patch which rendered the rings from tutorial 6. *Spreads*.

Connect the `Output` of the `Queue` to the `Y` input on the `Translate` node. and set the `LinearSpread` on the `Translate`'s `X` pin to have `Spread Count 20` and `Width 2`.

You should now have a graph of the past 20 values of sound amplitude.

12.3 FFT

The `FFT` gives us a detailed view of what intensity of each frequency is being heard by the computer.

To set it up, it's very similar to `RMS`. We create an `AudiIn`, set `Enabled` to `1`, and connect it to an `FFT` node.

To visualise its output, let's create an `IOBox` and attach it to the `FFT L` output of `FFT`. This gives 256 frequencies within the left channel. Low indexes in the spread are bass, high indices are treble. Since we have 256 values, it's hard to visualise it as numbers in an `IOBox`.

12.4 Visualising 256 slices in an IOBox

Go to the Inspektor and set the following properties for the `IOBox`

1. Set the `Columns` to `256`
2. Set `Show Value` to `false`
3. Set `Show Connections` to `true`
4. Set the `Maximum` to `1`
5. Set the `Minimum` to `0`

If the values are too small to see, then add an amplification factor between

12.5 Exercise

Try to recreate this patch.

Some notes:

- To fix the Aspect Ratio, we're using a **Scale** connected to the View transform pin on the **Renderer** rather than on the object. This applies the transform to the whole scene
- **Resample** is used to resize the spread from count 256 to **16**. The Mode is set to **Point** so that the values are resampled properly. This is like resampling an image in Photoshop
- A segment is an object that renders a circle or an arc. By Setting the Inner Radius to **0.95** we get a ring.